DSLM : Dynamic Synchronous Language with Memory

Pejman Attar

DSLM : Dynamic Synchronous Language with Memory*

Pejman Attar

INRIA Sophia Antipolis - Méditerranée
Pejman.Attar@inria.fr

Abstract. We propose a new language called DSLM based on the synchronous/reactive model. In DSLM, systems are composed of several sites executed asynchronously, while each site is running a number of agents in a synchronous way. An agent consists of a script and a memory. Scripts may call functions or modules which are handled in a host language. Two properties are assured by DSLM: reactivity of agents and absence of data-races between agents. In DSLM, we introduce a way to benefit from multi-core/multi-processor architectures by mapping each site to several cores, using a notion of synchronized scheduler.

1 Introduction

Concurrency and parallelism are among the main issues of systems and programming languages. Nowadays, multi-core machines are everywhere: in servers, PCs and even mobile phones. These machines are widely used by the public. Hence, concurrency and parallelism are no more specialist problems. They now also concern software programmers.

There exist several approaches to concurrent programming. In this paper, we are concerned with the shared memory concurrency model [12], the first and most used variant of which is called *preemptive multi-threading*. In this model, concurrent programs are threads that are scheduled and preempted by the system in an arbitrary way. The major problem of this model is the freedom of schedulers to arbitrarly choose the executed thread, which may lead to so-called *time-dependent errors*, which are generally considered as extremely difficult to tame [10].

Another popular variant is *cooperative multi-threading* [7] [19]. In this model, the system loses the possibility to arbitrarily preempt threads. To be given to a new thread, the control must be explicitly released by the currently executing thread. This rules out time-dependent errors. However, the cooperative approach suffers from a major drawback, as a single thread can freeze the whole system if it never releases the control.

An alternative way of dealing with concurrency and parallelism is that of the *synchronous approach*, which is based on event broadcast [14], [5]. We briefly

* with support from ANR-08-EMER-010, project PARTOUT

describe it by means of an example written in the *Esterel* language [4]. Consider a program made of two parallel statements, one waiting for an event $ev1$, then producing event $ev2$, and the other producing event $ev1$. Such a program is written in Esterel as: $P1$ = `await immediate ev1; emit ev2 || emit ev1`. This program emits both $ev1$ and $ev2$, and this is its only possible outcome. One of the standpoints of Esterel is that program execution should be deterministic.

However, synchronous parallelism has difficulties to cope with memory. Indeed, uncontrolled concurrent accesses to memory, as in: $P2$ = `x:=1 || x:=2`, may give rise to non-deterministic results: the outcome can be either $x = 1$ (if the second branch is executed first) or $x = 2$ otherwise. Moreover, non-determinism can result also from non-atomic access to the memory. Thus, parallel programming demands for restrictions on the use of the memory by the parallel components and for means to get atomicity in memory access. In the context of preemptive multi-threading, atomicity is usually provided by locks. Relying on locks for atomicity is problematic, however, because locks can produce deadlock situations or time-dependent errors, if incorrectly used [18].

In Esterel, the solution to these problems is rather drastic: a variable cannot be accessed by one branch of a parallel statement and written by the other. Thus, the previous program $P2$ would be rejected by the compiler. However, Esterel does not control concurrent accesses made at a lower level by procedures and functions. Consider the following statement, where two functions are called in parallel: $P3$ = `f1 () || f2 ()`. The Esterel compiler is unable to verify that no concurrent accesses occur through the calls of $f1$ and $f2$. Moreover, one may view the Esterel solution as over-restrictive, specially in the context of multi-core programming where access to the memory is the basic communication and synchronization means.

In this paper, we propose a new model inspired by both the cooperative and the synchronous approaches, in which data races and time-dependent errors are eliminated by construction. Our model is distributed and exhibits three kinds of parallelism: 1) asynchronous parallelism among sites, 2) synchronous parallelism among agents in a site and 3) synchronous cooperative parallelism among scripts within an agent. Moreover, this model appears to be well-suited to take benefit from multi-core/multi-processor architectures.

The paper is organized as follows. The model is introduced in Section 2. Section 3 presents the syntax. The semantics of scripts is given in Section 4. The semantics of sites and systems is presented in Section 5. A type system to verify safety of the language is given in Section 6. The intuition of how we can use this language to benefit from multi-core/multi-processor architectures is presented in Section 7. Finally Sections 8 and 9 discuss related work and give a conclusion. All results are given without proofs, which can be found in [3].

2 The Model

We propose a synchronous model which uses a deterministic asymmetric parallel operator (noted here by $|>$), and is able to deal with the memory in a safe way,

without possibilities of time-dependent errors or data-races. Our model is called `DSLM` to stand for *Dynamic Synchronous Language with Memory*.

As in standard synchronous models, a notion of instant is present. Instants define a logical time, different from the physical time; an instant is terminated when all the parallel components have reached a synchronization barrier. Our model makes sure that this synchronous barrier is actually reached, that is, it ensures the termination of instants.

In the synchronous model, events are used for synchronization. At each instant, an event is either absent, or present if it is produced during the instant. However, in standard synchronous languages like Esterel, *causality cycles* can appear when no coherent solution can be found for the absence/presence status of an event.

In addition to the issues of non-terminating instants and of causality cycles, the standard synchronous model has difficulties in facing real parallelism and making use of multi-core machines. Indeed, multi-core programming heavily relies on shared memory communication, which is excluded in these models. Moreover, standard synchronous languages do not allow dynamic creation of events and parallel components.

Our model is based on the reactive variant [2] of the synchronous approach, which allows for dynamic creation of events and agents, and avoids causality cycles by construction (by prohibiting instantaneous reaction to absence of events). Moreover, in this variant there exist solutions to insure the termination of instants by construction, even at the lower level of function and procedure calls (in the FunLoft [8] language).

The main ingredients of our model are scripts, events, agents and sites, shortly described below:

- Scripts : scripts are the basic parallel components.
- Events: events are instantaneously broadcast in our model. They may have associated values. Events and their associated values are seen by all components in the same way. Events can be created dynamically during execution.
- Agents: an agent encapsulates a script which can be made of several parallel components. These components share the agent's memory and they are the only ones that may access it.
- Sites: a site is a location where execution of agents takes place. Each site runs one or more agents and manages a set of events shared by these agents. There is no dynamic creation of sites. All agents belonging to the same site share the same instants and events. Agents can *migrate* from one site to another. A *system* is composed of a set of sites which run asynchronously. Agent migration is the only communication means between sites.

`DSLM` does not provide means to define functions. However, scripts may call functions or modules defined in a "host" language. Functions are required to terminate instantaneously (i.e. in the same instant they are started). By contrast, module execution can last several instants or even never terminate.

Note that our model can be considered as a member of the GALS family (Globally Asynchronous, Locally Synchronous) [17], as sites are executed asynchronously and agents in the same site are executed synchronously.

3 Syntax and Domains

In this section we introduce the syntax of DSLM.

The following disjoint countable sets are assumed: **LocName** (locations), **VarName** (variable names), **FunName** (function names), **ModuleName** (module names), **SiteName** (site names) and **EventName** (event names). Each set has an associated function which returns an unused element of the set (for example, each call of the function *new_loc* returns a new location in **LocName**).

We use the following notations to define domains: $A \times B$ denotes the cartesian product of the domains A and B; $A \oplus B$ denotes the disjoint union of the domains A and B; $\mathbb{N}^A$ denotes the set of multi-sets over A; $\uplus$ is the union of multi-sets; **None** is the domain that contains the unique distinguished element $None$; $\overrightarrow{A}$ denotes the domain of vectors over domain A; and $A \to B$ is the domain of (partial) functions from A to B.

Basic denotes the set of basic values, and **Value** the set that contains basic values as well as locations and vectors of values.

A memory M belonging to **Mem** is a partial function that associates a value with a location or variable. One notes $M[l \leftarrow v]$ the memory M' defined by: $M'(l) = v$ and for $x \neq l$, $M'(x) = M(x)$. If $M(x)$ is a location, $M[x \leftarrow v]$ is an abbreviation for $M[M(x) \leftarrow v]$.

Elements E of **EventEnv** are multi-sets of pairs composed of an event name and an associated basic value. For simplicity, we write $ev \in E$ if there exists v such that $(ev, v) \in E$. The function $get\_values(ev, E)$ is used to collect all the basic values associated with an event ev in a set of events E.

$b \in \mathbf{Basic} = \mathbf{Bool} \oplus \mathbf{Integer} \oplus \mathbf{Double} \oplus \mathbf{String}$

$v \in \mathbf{Value} = \mathbf{Basic} \oplus \mathbf{LocName} \oplus \overrightarrow{\mathbf{Value}}$

$M \in \mathbf{Mem} : (\mathbf{VarName} \oplus \mathbf{LocName}) \to \mathbf{Value}$

$E \in \mathbf{EventEnv} = \mathbb{N}^{(\mathbf{EventName} \times \mathbf{Basic})}$

$get\_values : \mathbf{EventName} \times \mathbf{EventEnv} \to \mathbb{N}^{\mathbf{Basic}}$

The syntax of expressions and scripts is defined as follow:

$e \in \mathbf{Expr} ::= v \mid x \mid !\,x \mid \overrightarrow{e} \mid \texttt{ref}\ e \mid f(e)$

$s \in \mathbf{Script} ::=$ `nothing` $\mid s; s \mid x := e$
$\mid s \mid\!> s \mid$ `let` $x = e$ `in` s `end` $\mid$ `cooperate`
$\mid$ `generate` $(ev, e) \mid$ `await` $ev \mid$ `get_all` ev `in` l
$\mid$ `do` s `watching` $ev \mid$ `repeat` e `do` s `end` $\mid$ `loop` s `end`
$\mid$ `launch` $m(e) \mid$ `if` e `then` s `else` s `end` $\mid$ `migrate to` $site$
$\mid$ `createAgent` s `in` $site$

The set **Agent** consists of agents, which are triples of the form (s, M, η), where $s \in \mathbf{Script}$, $M \in \mathbf{Mem}$, and η is a migration request which may be of two

forms: $None$, indicating the absence of migration request, and $site$, expressing a demand for migrating the current agent to site $site$.

$Ag \in \mathbf{Agent} = \mathbf{Script} \times \mathbf{Mem} \times \mathbf{Migr}$
$\eta \in \mathbf{Migr} = None \oplus \mathbf{SiteName}$
$S \in \mathbf{Site} = \mathbf{SiteName} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbf{EventEnv}$

Finally, **Site** is the set of sites. The site $(site, \mathcal{A}, \mathcal{I}, E)$ is interpreted as follows:

- $site$ is the name of the site;
- $\mathcal{A}$ is the multi-set of the agents running in the site;
- $\mathcal{I}$ is the multi-set of incoming agents, which will be incorporated in the site at the next instant.
- E is the multi-set of events associated with their values generated in the site in the current instant.

Let $S = (site, \mathcal{A}, \mathcal{I}, E)$ be a site; one notes $sn(S) \subseteq \mathbf{SiteName}$ the set of site names occurring in $\mathcal{A}$.

The set **Sys** consists of systems, which are sets of sites $\Sigma = \{S_1, ..., S_n\}$. A system $\Sigma = \{S_1, ..., S_n\}$ where $S_i = (site_i, \mathcal{A}_i, \mathcal{I}_i, E_i)$ is *well-formed* if the following two requirements are fulfilled:

- Sites have distinct names: $\forall i, j \in \{1, \ldots, n\} : \quad i \neq j \;\Rightarrow\; site_i \neq site_j$.
- Migration targets exist: $\forall i \in \{1, \ldots, n\} : site \in sn(S_i) \;\Rightarrow\; \exists j \,.\, site = site_j$.

A reconstruction function $\Omega : \mathbf{Agent} \times \mathbf{EventEnv} \to \mathbf{Agent}$, defined in Section 5.3, is used to prepare an agent for the next instant.

4 Semantics of Scripts

This section presents the semantics of scripts. We start by defining the semantics of expressions. Then, we introduce the suspension predicate for scripts. Finally, we give the small-step semantics of scripts.

4.1 Expressions

The evaluation of an expression is noted $e, M \rightsquigarrow v, M'$, where e is the initial expression, M is the memory in which e is evaluated, v is the result of the evaluation of e, and M' is the new memory after the evaluation.

$$v, M \rightsquigarrow v, M \ (val) \qquad !\,x, M \rightsquigarrow M(x), M \ (access)$$

$$\frac{e_i, M_i \rightsquigarrow v_i, M_{i+1}}{(e_1, ..., e_n), M_1 \rightsquigarrow (v_1, ..., v_n), M_{n+1}} \ (vec) \qquad \frac{e, M \rightsquigarrow v, M' \quad l = new\_loc()}{\texttt{ref}\ e, M \rightsquigarrow l, M'[l \leftarrow v]} \ (ref)$$

$$\frac{\overrightarrow{e}, M \rightsquigarrow \overrightarrow{v}, M' \quad f(\overrightarrow{v}) = v'}{f(\overrightarrow{e}), M \rightsquigarrow v', M'} \ (fun)$$

Evaluation of an expression is defined above. Rules *val* and *access* are standard. The elements of a vector of expressions are evaluated in increasing order (Rule *vec*). Evaluation of `ref` e returns a new location in which the value of e is stored (Rule *ref*). The evaluation of a function call is assumed to be instantaneous. The only changes in the memory are the ones resulting from the evaluation of function arguments (Rule *fun*).

4.2 Suspension Predicate

Reactive programs suspend execution either waiting for signals to be produced, or waiting for the end of the current instant. One writes $\langle s, E \rangle \ddagger$ to indicate that the script s is suspended in the environment E. The *suspension predicate* $\ddagger$ is defined inductively by the rules given below.

The `cooperate` and `get_all` instructions are suspended in all environments (Rules *coop*, *get_all*). An `await` instruction is suspended when the awaited event is not present (Rule $await_0$). A watching statement is suspended if its body is suspended (Rule $watch_0$). A sequence is suspended if its first components is suspended (Rule seq_0). A parallel statement is suspended if the first components is suspended and the second one is either suspended or terminated (Rule $para_0$).

$$\langle \texttt{cooperate}, E \rangle \ddagger \ (coop) \qquad \langle \texttt{get\_all}\ ev\ \texttt{in}\ l, E \rangle \ddagger \ (get\_all)$$

$$\frac{ev \notin E}{\langle \texttt{await}\ ev, E \rangle \ddagger}\ (await_0) \qquad \frac{\langle s, E \rangle \ddagger}{\langle \texttt{do}\ s\ \texttt{watching}\ ev, E \rangle \ddagger}\ (watch_0)$$

$$\frac{\langle s_1, E \rangle \ddagger}{\langle s_1; s_2, E \rangle \ddagger}\ (seq_0) \qquad \frac{\langle s_1, E \rangle \ddagger \qquad \langle s_2, E \rangle \ddagger \quad or \quad s_2 = \texttt{nothing}}{\langle s_1 |> s_2, E \rangle \ddagger}\ (para_0)$$

4.3 Transition Relation

The small-step semantics of scripts is relative to an event environment and a memory. It is given in terms of labeled transitions whose label is a *drop order*. Drop orders record the migration requests issued by migrate instructions. A drop order $d \in \mathbf{D}$ may be of three forms: 1) $site \in \mathbf{SiteName}$ is a demand for the migration of the current agent to $site$, 2) $(Ag, site) \in \mathbf{Agent} \times \mathbf{SiteName}$ is a demand for the migration to $site$ of the newly created agent Ag, 3) $None$ is the absence of migration request.

The general format of a script transition is $\langle s,\ E,\ M \rangle \xrightarrow{d} \langle s',\ E',\ M' \rangle$ where:

- s is the script which is rewritten;
- E is a multi-set of pairs (ev, v);
- s' is the *residual* script (what remains to be done);
- $E' = E \cup \{(ev, v)\}$ if event ev is generated with value v during the step, and $E' = E$ otherwise;
- M is the memory before the rewriting of s;
- M' is the new memory obtained after the rewriting of s;
- d is a drop order reflecting the migration request issued from the rewriting.

$$\frac{\langle s_1,\, E,\, M\rangle \xrightarrow{d} \langle s_1',\, E',\, M'\rangle}{\langle s_1; s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_1'; s_2,\, E',\, M'\rangle}\,(seq_1) \quad \langle \texttt{nothing}; s_2,\, E,\, M\rangle \xrightarrow{None} \langle s_2,\, E,\, M\rangle\,(seq_2)$$

$$\frac{\langle s_1,\, E,\, M\rangle \xrightarrow{d} \langle s_1',\, E',\, M'\rangle}{\langle s_1 |\!>\! s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_1' |\!>\! s_2,\, E',\, M'\rangle}\,(para_1) \quad \langle \texttt{nothing} |\!>\! s_2,\, E,\, M\rangle \xrightarrow{None} \langle s_2,\, E,\, M\rangle\,(para_2)$$

$$\frac{\langle s_1, E\,\rangle \ddagger \qquad \langle s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_2',\, E',\, M'\rangle}{\langle s_1 |\!>\! s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_1 |\!>\! s_2',\, E',\, M'\rangle}\,(para_3)$$

$$\frac{e, M \rightsquigarrow v, M'}{\langle \texttt{let}\ x = e\ \texttt{in}\ s\ \texttt{end},\, E,\, M\rangle \xrightarrow{None} \langle s,\, E,\, M'[x \leftarrow v]\rangle}\,(let)$$

$$\frac{e, M \rightsquigarrow v, M'}{\langle x := e,\, E,\, M\rangle \xrightarrow{None} \langle \texttt{nothing},\, E,\, M'[x \leftarrow v]\rangle}\,(assign)$$

$$\frac{\langle s |\!>\! \texttt{cooperate},\, E,\, M\rangle \xrightarrow{d} \langle s',\, E',\, M'\rangle}{\langle \texttt{loop}\ s\ \texttt{end},\, E,\, M\rangle \xrightarrow{d} \langle s'; \texttt{loop}\ s\ \texttt{end},\, E',\, M'\rangle}\,(loop)$$

$$\frac{e, M \rightsquigarrow v, M' \quad E' = E \uplus \{(ev, v)\}}{\langle \texttt{generate}\ (ev, e),\, E,\, M\rangle \xrightarrow{None} \langle \texttt{nothing},\, E',\, M'\rangle}\,(gen)$$

$$\frac{ev \in E}{\langle \texttt{await}\ ev,\, E,\, M\rangle \xrightarrow{None} \langle \texttt{nothing},\, E,\, M\rangle}\,(await)$$

$$\frac{\langle s,\, E,\, M\rangle \xrightarrow{d} \langle s',\, E',\, M'\rangle}{\langle \texttt{do}\ s\ \texttt{watching}\ ev,\, E,\, M\rangle \xrightarrow{d} \langle \texttt{do}\ s'\ \texttt{watching}\ ev,\, E',\, M'\rangle}\,(watch_1)$$

$$\langle \texttt{do nothing watching}\ ev,\, E,\, M\rangle \xrightarrow{None} \langle \texttt{cooperate},\, E,\, M\rangle\,(watch_2)$$

$$\frac{\overrightarrow{e}, M \rightsquigarrow \overrightarrow{v}, M' \quad ev = new\_event() \quad m(\overrightarrow{v}, ev) \Uparrow}{\langle \texttt{launch}\ m\,(\overrightarrow{e}),\, E,\, M\rangle \xrightarrow{None} \langle \texttt{await}\ ev,\, E,\, M'\rangle}\,(launch)$$

$$\frac{e, M \rightsquigarrow n, M'}{\langle \texttt{repeat}\ e\ \texttt{do}\ s\ \texttt{end},\, E,\, M\rangle \xrightarrow{None} \langle \overbrace{s; \ldots; s}^{n\ times},\, E,\, M'\rangle}\,(repeat)$$

$$\frac{e, M \rightsquigarrow tt, M'}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end},\, E,\, M\rangle \xrightarrow{None} \langle s_1,\, E,\, M'\rangle}\,(if)$$

$$\langle \texttt{migrate to}\ site,\, E,\, M\rangle \xrightarrow{site} \langle \texttt{cooperate},\, E,\, M\rangle\,(selfmigr)$$

$$\langle \texttt{createAgent}\ s\ \texttt{in}\ site,\, E,\, M\rangle \xrightarrow{(s,\emptyset,None)\downarrow site} \langle \texttt{nothing},\, E,\, M\rangle\,(creatAg)$$

Fig. 1. Script Semantics

The semantics of scripts is given in Fig.1. Let us briefly explain the rules.

The semantics of sequence, conditional, let and assignment are standard. Execution of a parallel instruction always starts by rewriting the left branch (Rule $para_1$). Once the left branch is over (`nothing`), the script reduces to its right branch (Rule $para_2$). If the left branch is suspended, the execution of the right branch starts (Rule $para_3$). We note that we are using an *immediate cooperation*, which means that the right branch execution stops as soon as the left branch is no more suspended. A loop statement executes its body cyclically. A cooperate statement is systematically added in parallel to the body to avoid instantaneous loops (Rule *loop*). A generate instruction produces an event and associated with the value obtained from the evaluation of an expression (Rule *generate*). The pair made of the event and its value is added in the event environment. Several productions of the same event with the same value are possible during the same instant. An `await` instruction terminates if the awaited event is present in the environment (Rule *await*); there is no rule corresponding to an event which is not present. In this case the instruction is suspended: $\langle \texttt{await}\ ev, E \rangle \ddagger$. A `watching` statement executes its body, if not terminated, and rewrites to a new watching statement (Rule $watch_1$). If the body is terminated (i.e. it is `nothing`), then the `watching` statement rewrites to a `cooperate` instruction (Rule $watch_2$). A module call launches a new separate thread run by the operating system to execute a specific module. Execution of a module can take several instants or never terminates. The thread executing a module is supposed to generate a termination event when the module terminates (Rule *launch*). A `repeat` statement executes its body in sequence n times, where n is the integer obtained by evaluating an expression (Rule *repeat*). An agent creation produces a drop order $Ag \downarrow site$ to demand the migration in *site* of a new agent Ag containing a script s and a new empty memory $\emptyset$ (Rule $creatAg$).The absorption of the newly created agent by the system will be described in Rule eoi_2. A migration instruction *site* produces a drop order *site* to demand the migration in *site* of the executing agent, and suspends up to the end of the current instant (Rule $selfmigr$). The processing of a drop order will be described in Rule $site_3$.

The transition relation thus defined satisfies the following properties:

Proposition 1. *(Determinism)* $\forall s \in$ **Script** *: if* $\langle s, E, M\rangle \xrightarrow{d_1} \langle s_1, E_1, M_1\rangle$ *and* $\langle s, E, M\rangle \xrightarrow{d_2} \langle s_2, E_2, M_2\rangle$ *then* $d_1 = d_2$ *and* $\langle s_1, E_1, M_1\rangle = \langle s_2, E_2, M_2\rangle$.

Proposition 2. *(Reactivity)*
The execution of every script is bounded in any memory and event environment:

$\forall s \in$ **Script**, $\forall E \in$ **EventEnv**, $\forall M \in$ **Mem**, $\exists n.\ \ \langle s, E, M\rangle \xrightarrow{d_1} \ ... \ \xrightarrow{d_n} \langle s_n, E_n, M_n\rangle \wedge \ (s_n = \texttt{nothing} \vee \langle s_n, E_n \rangle \ddagger\ )$

5 Semantics of Sites and Systems

We give now the small-step semantics of sites and systems. The execution of a site during one instant is described by the first three Rules sys_1- sys_3. The

next two rules deal with the end of instants. Migration requests are processed in Rule eoi_1. The transition to the next instant is described by Rule eoi_2. Finally, we define the transformation of the suspended terms for the next instant.

5.1 Sites

The format for system rewriting is $\Sigma \to \Sigma'$. It is specified by three rules, which describe the choice of a site S within Σ, the choice of an agent Ag in S, and the execution of Ag in the event environment of S. We use a function to deal with migration requests noted $\blacktriangleright$. This function handles a sequence of migration requests for a given agent by selecting the first one and ignoring the others.

$$\blacktriangleright\colon \mathbf{Migr} \times \mathbf{SiteName} \to \mathbf{SiteName} \qquad \eta \blacktriangleright site = \begin{cases} site & if\ \eta = None \\ \eta & otherwise \end{cases}$$

The Rule sys_1 considers the case where no drop order is issued from the agent execution. After execution, the agent is reintegrated in the site and the site event environment is updated. The second Rule sys_2 corresponds to the production of the drop order for a new agent Ag in $site_0$. Agent Ag is put in the set of incoming agents for $site_0$. The Rule sys_3 corresponds to the production of a migration request $site_0$ for the current agent. There are two cases: either a migration request is already present in the agent, and then $site_0$ is simply ignored (a way to prevent schizophrenia...); or, there is no previous migration request in the agent, and then $site_0$ becomes the migration request of the agent.

$$\frac{S = (site, \mathcal{A} \uplus (s, M, \eta), \mathcal{I}, E) \quad \langle s,\ E,\ M\rangle \xrightarrow{None} \langle s',\ E',\ M'\rangle}{\Sigma \cup S \quad \to \quad \Sigma \cup (site, \mathcal{A} \uplus (s', M', \eta), \mathcal{I}, E')}\ (sys_1)$$

$$\frac{\begin{array}{c} S = (site, \mathcal{A} \uplus (s, M, \eta), \mathcal{I}, E) \quad \langle s,\ E,\ M\rangle \xrightarrow{Ag \downarrow site_0} \langle s',\ E',\ M'\rangle \\ S'_0 = (site_0, \mathcal{A}_0, \mathcal{I}_0 \uplus Ag, E_0) \quad S' = (site, \mathcal{A} \uplus (s', M', \eta), \mathcal{I}, E') \end{array}}{\Sigma \cup (site_0, \mathcal{A}_0, \mathcal{I}_0, E_0) \cup S \to \Sigma \cup S'_0 \cup S'}\ (sys_2)$$

$$\frac{S = (site, \mathcal{A} \uplus (s, M, \eta), \mathcal{I}, E) \quad \langle s,\ E,\ M\rangle \xrightarrow{site_0} \langle s',\ E',\ M'\rangle}{\Sigma \cup S \quad \to \quad \Sigma \cup (site, \mathcal{A} \uplus (s', M', \eta \blacktriangleright site_0), \mathcal{I}, E')}\ (sys_3)$$

5.2 End of Instants

The suspension predicate of scripts is extended to agents: an agent is suspended if its script is suspended (Rule *agent*). A site $S = (site, \mathcal{A}, \mathcal{I}, E)$ is suspended if all its agents are suspended or terminated (Rule *site*). The predicate $\natural$ indicates the absence of migration requests in a site (Rule *nomigr*). The end of instant is detected when no migration request is left on the site and all the agents are suspended; then the site moves to the next instant, as described by Rule eoi_1.

$$\frac{\langle s, E\ \rangle\ \ddagger}{\langle (s, M, \eta), E\ \rangle\ \ddagger}\ (agent) \qquad \frac{\forall (s, M, \eta) \in \mathcal{A} \quad \eta = None}{(site, \mathcal{A}, \mathcal{I}, E)\natural}\ (nomigr)$$

$$\frac{\forall Ag \in \mathcal{A} \quad \langle Ag, E \rangle \ddagger \qquad \vee \qquad Ag = (\texttt{nothing}, M, \eta)}{(site, \mathcal{A}, \mathcal{I}, E) \ddagger} \; (site)$$

When a site is suspended, that is when all its agents are suspended, the current instant terminates, and a new instant can start.

Two rules are needed to process suspended sites. The first one considers migration requests of existing agents to a different site, while the second incorporates new (incoming) agents into the requested site. In both cases, suspended agents are transformed to take into account the absence of events. These two transformations are defined using the function Ω described in Section 5.3.

– The first rule treats the case where an agent Ag_1 of a suspended site $site_1$ requests to migrate to site $site_2$. First, suspended scripts of Ag_1 are processed by Ω; then the resulting agent is added to the set of agents of $site_2$:

$$\frac{\begin{array}{c} S_1 \ddagger \\ S_1 = (site_1, \mathcal{A}_1 \uplus (s, M, site_2), \mathcal{I}_1, E_1) \quad S_2 = (site_2, \mathcal{A}_2, \mathcal{I}_2, E_2) \\ S_1' = (site_1, \mathcal{A}_1, \mathcal{I}_1, E_1) \quad S_2' = (site_2, \mathcal{A}_2 \uplus \Omega\,((s, M, \eta), E_1), \mathcal{I}_2, E_2) \end{array}}{\Sigma \cup S_1 \cup S_2 \quad \rightarrow \quad \Sigma \cup S_1' \cup S_2'} \; (eoi_1)$$

– The second rule considers the case where there is no migration request ($\hookrightarrow$ denotes the passing of the site to the next instant). In this case, suspended instructions are processed by function Ω, and the agents requesting to be incorporated in the site (the incoming agents) are added to the agent set. Moreover, the site event environment is reset to $\emptyset$:

$$\frac{S \ddagger, S\natural \quad S = (site, \mathcal{A}, \mathcal{I}, E)}{\Sigma \cup S \quad \hookrightarrow \quad \Sigma \cup (site, \Omega\,(\mathcal{A}, E) \uplus \mathcal{I}, \emptyset, \emptyset)} \; (eoi_2)$$

In the above rule, $\Omega\,(\mathcal{A}, E)$ means $\{\Omega\,(Ag, E) \mid Ag \in \mathcal{A}\}$.

5.3 Reconstruction for Next Instant

The reconstruction function Ω is used at each end of instant in order to reconstruct suspended agents, with regard to an event environment E, and to prepare them for execution at the next instant. To reconstruct an agent means to clean-off its script and this reconstruction can possibly modify the agent's memory. The Ω function is first defined inductively on scripts by:

$$\Omega\,(\texttt{cooperate}, E, M) = (\texttt{nothing}, M)$$

$$\Omega\,(\texttt{get\_all}\ ev\ \texttt{in}\ l, E, M) = (\texttt{nothing}, M[l \leftarrow get\_values(ev, E)])$$

$$\frac{ev \in E}{\Omega\,(\texttt{do}\ s\ \texttt{watching}\ ev, E, M) = (\texttt{nothing}, M)}$$

$$\frac{ev \notin E \quad \Omega\,(s, E, M) = (s', M')}{\Omega\,(\texttt{do}\ s\ \texttt{watching}\ ev, E, M) = (\texttt{do}\ s'\ \texttt{watching}\ ev, M')}$$

$$\Omega\,(\texttt{await}\ ev, E, M) = (\texttt{await}\ ev, M)$$

$$\frac{\Omega\,(s_1, E, M) = (s_1', M_1)}{\Omega\,(s_1; s_2, E, M) = (s_1'; s_2, M_1)}$$

$$\frac{\Omega\,(s_1, E, M) = (s_1', M_1) \quad \Omega\,(s_2, E, M_1) = (s_2', M_2)}{\Omega\,(s_1 |> s_2, E, M) = (s_1' |> s_2', M_2)}$$

There are four basic cases for script reconstruction:

- `cooperate` is reconstructed in `nothing`;
- `do` s `watching` ev is reconstructed in `nothing` if $ev \in E$; otherwise, it is reconstructed in `do` s' `watching` ev where s' is the reconstruction of s in E.
- `await` ev is reconstructed in itself;
- `get_all` ev `in` l is reconstructed in `nothing`; moreover, the values associated with ev in E are collected in a list which is assigned to l. Note that this is the only reconstruction step that possibly modifies the memory.

The Ω function is extended to agents as follows: $\Omega\,((s, M, \eta), E) = (s', M', \eta)$ if $\Omega\,(s, E, M) = (s', M')$.

6 Type system

The purpose of the proposed type system for scripts is twofold: first, to insure that values are correctly used, as in traditional type checking, to verify for instance that in `if` e `then` s_1 `else` s_2 `end`, e is a Boolean expression; second, to ensure that no data-race occurs. For example, consider the following script:

```
let x = ref e1 in createAgent !x in remote; x := e2 end
```

There is a data-race as `x` is read by an agent belonging to site `remote`, while it is written in the current site. To prevent this kind of errors, the type system checks that a reference not belonging to an agent memory (that is, not created in the agent) cannot be accessed by the agent.

A *type* is either the name of a basic type (*int, bool,* etc), the empty type (`unit`) or a reference on a *type*:

$$Basic ::= \texttt{unit} \,|\, bool \,|\, int \,|\, string \qquad \tau ::= Basic \,|\, ref\ \tau \,|\, \overrightarrow{\tau}$$

A typing environment Γ is a possibly empty set[1] of elements of the form $x : \tau$, where x is a variable and τ is a type: $\Gamma ::= x_1 : \tau_1, \cdots, x_n : \tau_n$. The general form of typing judgments is: $\Gamma \vdash s : \tau$ where Γ is the typing environment, s is the script to be typed, and τ is the type of s in Γ.

[1] In the sequel, the brackets of the standard set notation are omitted.

6.1 Typing Rules

To be typed, a variable must be present in the typing environment (Rule var_t). To type a sequence or a parallel, both branches must be typed (Rule seq_t, $para_t$). A conditional is typed if its expression is a Boolean and its two branches are typed (Rule if_t). A `repeat`, `loop` or `watching` statements is typed by checking that the body is typed. In the `repeat`, the expression should be an integer (Rule $repeat_t$, $loop_t$, $watch_t$). An agent creation statement is typed by checking its body in an empty environment (Rule $creatAg_t$). This is the central rule to prevent the possibility of data-races. The type system rules are:

$$\Gamma \cup \{x : \tau\} \vdash x : \tau \ (var_t) \quad \Gamma \vdash v : \tau \ (val_t) \quad \frac{\Gamma \vdash x : \texttt{ref}\ \tau}{\Gamma \vdash !\, x : \tau}(access_t)$$

$$\frac{\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \quad f : \overrightarrow{\tau} \rightarrow \tau'}{\Gamma \vdash f(\overrightarrow{e}) : \tau'}\ (fun_t) \quad \frac{\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \quad m : \overrightarrow{\tau} \rightarrow \texttt{unit}}{\Gamma \vdash \texttt{launch}\ m\,(\overrightarrow{e}) : \texttt{unit}}\ (vec_t)$$

$$\frac{\Gamma \vdash s_1 : \texttt{unit} \quad \Gamma \vdash s_2 : \texttt{unit}}{\Gamma \vdash s_1\ ;\ s_2 : \texttt{unit}}\ (seq_t) \quad \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \cup x{:}\ \tau_1 \vdash s : \texttt{unit}}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ s\ \texttt{end} : \texttt{unit}}(let_t)$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash s_1 : \texttt{unit} \quad \Gamma \vdash s_2 : \texttt{unit}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} : \texttt{unit}}\ (if_t) \quad \frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash s : \texttt{unit}}{\Gamma \vdash \texttt{repeat}\ e\ \texttt{do}\ s : \texttt{unit}}\ (repeat_t)$$

$$\frac{\Gamma \vdash s : \texttt{unit}}{\Gamma \vdash \texttt{loop}\ s : \texttt{unit}}\ (loop_t) \quad \frac{\Gamma \vdash s_1 : \texttt{unit} \quad \Gamma \vdash s_2 : \texttt{unit}}{\Gamma \vdash s_1 |> s_2 : \texttt{unit}}\ (para_t)$$

$$\Gamma \vdash \texttt{cooperate} : \texttt{unit}\ (coop_t)$$

$$\Gamma \vdash \texttt{await}\ ev : \texttt{unit}\ (await_t) \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash x : \texttt{ref}\ \tau}{\Gamma \vdash x := e : \texttt{unit}}(set_t)$$

$$\frac{\Gamma \vdash e : Basic}{\Gamma \vdash \texttt{generate}\ (ev, e) : \texttt{unit}}\ (gen_t) \quad \frac{\Gamma \vdash s : \texttt{unit}}{\Gamma \vdash \texttt{do}\ s\ \texttt{watching}\ ev : \texttt{unit}}\ (watch_t)$$

$$\Gamma \vdash \texttt{get\_all}\ ev\ \texttt{in}\ l : \texttt{unit}\ (get\_all_t) \quad \frac{\emptyset \vdash s : \texttt{unit}}{\Gamma \vdash \texttt{createAgent}\ s\ \texttt{in}\ site : \texttt{unit}}\ (creatAg_t)$$

$$\Gamma \vdash \texttt{migrate to}\ site : \texttt{unit}\ (self\_migr_t)$$

The following proposition holds:

Proposition 3. *(Safety) If a program P is well-typed, then no data-race can occur during its execution.*

7 Proposal for Multi-core Architectures

We turn now to the context of multi-processor/multi-core architectures. Our main goal is to give the system the possibility to maximize the usage of computing resources (processors or cores).

We introduce in the model a new level of parallelism in which the agents are mapped to parallel components called *synchronized schedulers* (or for simplicity, *schedulers*). Sites are composed of several schedulers which share the same instants and the same events.

At implementation level, the intention is that each scheduler is executed by a distinct thread (for example, in a Linux-SMP architecture), or by a distinct processor (for example, in a cluster). Schedulers within a site are supposed to run in real parallelism and to synchronize at the end of each instant (via a synchronization barrier). The number of synchronized schedulers belonging to a given site dynamically changes during the site execution, according to the load of agents that are present on the site and to the availability of computing resources. Moreover, agents can be *transparently redistributed* among schedulers of the same site, to balance the charge of agents over the sites. Transparency basically results from the fact that agents do not share memory. Actually, the only way to share information in a site is to use events.

According to this approach, the assignment of agents to schedulers is not statically fixed, in order to allow the system to use resources in an efficient way. For example, in a multi-core context, the system is free to optimize the mapping of schedulers to cores (and consequently the mapping of agents) in a way that maximizes the use of the real cores. Initially, one scheduler, arbitrarily chosen, is associated with each site. The remaining schedulers, if any, are the *unused schedulers*. At run time, two actions are possible for a site: first, the activation of an unused scheduler (which thus becomes used); second, the releasing of a scheduler (which becomes unused). The first action is called *site expansion* and the second *site contraction*. The conditions for performing expansions and contractions are not specified and are left to the implementation, in order to maximize the possibilities of optimizations.

The way sites are structured in sets of synchronized schedulers is formalized through a semantics described in [3]. This reference also includes an example (colliding particles).

8 Related Work

We discuss now some related work and compare it with our approach.

Ptolemy [13] is a complex framework which aims to model, design, and simulate concurrent, real-time embedded systems. Safety is not a central objective of Ptolemy. By contrast, `DSLM` is just a language, which focuses on safety, reactivity, distribution and maximal usage of computing resources.

SugarCubes [9] is a framework for reactive programming in Java. `DSLM` is strongly inspired by SugarCubes. Both formalisms use a similar totally deterministic parallel operator, called *merge* in SugarCubes. However, SugarCubes does not possess the notion of a synchronized scheduler and is not optimized for systems with multiple computing resources. SugarCubes can be used over the network by Java (RMI), which is not yet possible in our language.

FunLoft [8] (*"Functional Language over Fair Threads"*) is a language for safe reactive programming, with type inference. The FunLoft compiler checks that functions called by a program always terminate and only use a bounded amount of resources (memory and CPU). The theoretical basis of FunLoft is described in [11]. `DSLM` is strongly linked to FunLoft in two aspects: first, the notion of synchronized scheduler of DSLM comes from FunLoft. Second, there exists an experimental implementation of `DSLM` in FunLoft ([1]) in which functions are proved to terminate instantaneously. `DSLM` improves on FunLoft by introducing the possibility of dynamic load balancing of agents among synchronized schedulers inside a same site.

`ReactiveML` [15] is a language for reactive programming in ML. As ML, `ReactiveML` is safe in the sense that there is no possibility of a crash during execution. However, in `ReactiveML` there is no instant termination check. On the other hand, `ReactiveML`, like ML, is not presently adapted for multi-core architectures. `ReactiveML` offers the possibility to compile programs on the fly, which is not currently possible in `DSLM`.

ULM [6] is a model addressing the unreliable character of resource access in a global computing context. Like `DSLM`, ULM tries to achieve safety in memory access without using locks. However, in ULM, a script that wants to access a memory location is suspended if the location does not belong to the current site. ULM is not currently adapted for multi-core architectures.

9 Conclusion

We have presented an approach to dynamic parallel programming, based on the synchronous/reactive model. Our proposal aims at answering the following questions: (1) how to be sure that the program is indeed reactive? (2) how to avoid harmful interferences between parallel computations (e.g. data-races)? (3) how to execute programs efficiently on a multi-core/multi-processor architecture? We insure the reactivity of a program by construction, and we require functions to be instantaneous. In the current implementation, which basically translates `DSLM` in FunLoft, this property is checked by the FunLoft compiler. `DSLM` defines agents which encapsulate their memory in a way which forbids harmful interferences.

We envision the following tracks for future work:

1. Extend `DSLM` with security features such as access control and secure information flow, building on previous work for a core reactive language [16].
2. Integrate functions and modules in `DSLM`. Presently, functions and modules are defined in the host language, thus there is no insurance that the required properties (instantaneous termination of functions and non-instantaneous execution of modules) are satisfied. Having them directly defined in `DSLM` would allow us to statically check these properties.
3. Investigate the possibility to give a big-step semantics to agents and sites (but not at the script level). This semantics would give a more abstract view of systems and sites. It is made possible by the confluent character of parallelism within sites.

4. Complete the implementation of `DSLM`, which is currently under development, based on a translation to FunLoft.

In this perspective, `DSLM` would constitute a complete proposal for a safe and secure parallel programming language, adapted to multi-core/multi-processor architectures. This would be, to our knowledge, something new.

References

1. Partout Project Site: https://gforge.inria.fr/projects/partout/.
2. Reactive Programming Site. http://www-sop.inria.fr/indes/rp/.
3. Pejman Attar. DSLM : Dynamic Synchronous Language with Memory. Technical report, November 2012.
4. G. Berry. The Constructive Semantics of Pure Esterel, 1999.
5. G. Berry and L. Cosserat. The ESTEREL Synchronous Programming Language and Its Mathematical Semantics. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448.
6. G. Boudol. ULM: A Core Programming Model for Global Computing. In D. Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. 2004.
7. G. Boudol. Fair Cooperative Multithreading. In *CONCUR 2007 Concurrency Theory*, volume 4703, pages 272–286. 2007.
8. F. Boussinot. *Safe Reactive Programming: The FunLoft Proposal.* Lambert Academic Publishing, 2010.
9. F. Boussinot and J-F. Susini. The SugarCubes Tool Box: A Reactive Java Framework. *Software: Practice and Experience*, 28(14):1531–1550, 1998.
10. P. Brinch Hansen. Concurrent Programming Concepts. *ACM Comput. Surv.*, 5(4):223–245, December 1973.
11. F. Dabrowski. Programmation Réactive Synchrone: langages et contrôle des ressources. 2007. PhD thesis.
12. M. Dubois and C. Scheurich. *Software Engineering, IEEE Transactions on Memory Access Dependencies in Shared-memory Multiprocessors*, 16(6):660 –673, jun 1990.
13. J. Eker, J.W. Janneck, E.A. Lee, L. Jie, L. Xiaojun, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong X. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91:127 – 144, 2003.
14. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Pub., 1993.
15. L. Mandel and M. Pouzet. ReactiveML: A Reactive Extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 82–93. ACM, 2005.
16. A. Almeida Matos, G. Boudol, and I. Castellani. Typing noninterference for Reactive Programs. *J. Log. Algebr. Program.*, 72(2):124–156, 2007.
17. J. Muttersbach, T. Villiger, and W. Fichtner. Practical Design of Globally-Asynchronous Locally-synchronous Systems. In *Advanced Research in Asynchronous Circuits and Systems*, pages 52 –59, 2000.
18. A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
19. P. Schaumont, Bo-Cheng C. L., Wei Q., and I. Verbauwhede. Cooperative Multithreading on Embedded Multiprocessor Architectures Enables Energy-scalable Design. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 27 – 30, 2005.